Java Access Modifiers, Encapsulation & Getter-Setter Methods

1. Access Modifiers in Detail

Access modifiers control the visibility and accessibility of classes, methods, constructors, and variables in Java.

Types of Access Modifiers

1. public

- **Visibility**: Accessible from anywhere in the program
- **Usage**: Classes, methods, constructors, variables
- **Scope**: No restrictions

```java
public class PublicExample {
    public String publicVariable = "I'm accessible everywhere";

    public void publicMethod() {
        System.out.println("This method is accessible from anywhere");
    }

    public PublicExample() {
        System.out.println("Public constructor");
    }
}

// Can be accessed from any package
class TestPublic {
    public static void main(String[] args) {
        PublicExample obj = new PublicExample();
        System.out.println(obj.publicVariable);  // Accessible
        obj.publicMethod();                      // Accessible
    }
}
```

2. private

- **Visibility**: Accessible only within the same class
- **Usage**: Methods, constructors, variables (not top-level classes)
- **Scope**: Same class only

```java
```

```java
public class PrivateExample {
    private String privateVariable = "Only accessible within this class";
    private int secretNumber = 42;

    private void privateMethod() {
        System.out.println("This method is only accessible within this class");
    }

    private PrivateExample() {
        System.out.println("Private constructor");
    }

    // Public method to access private members
    public void accessPrivateMembers() {
        System.out.println(privateVariable);    // Accessible within same class
        privateMethod();                        // Accessible within same class
    }

    public static PrivateExample createInstance() {
        return new PrivateExample();            // Private constructor accessible within same class
    }
}

class TestPrivate {
    public static void main(String[] args) {
        PrivateExample obj = PrivateExample.createInstance();
        // System.out.println(obj.privateVariable);  // Compilation Error
        // obj.privateMethod();                      // Compilation Error
        obj.accessPrivateMembers();                  // Works - public method
    }
}
```

3. protected

- **Visibility**: Accessible within the same package and in subclasses (even in different packages)
- **Usage**: Methods, constructors, variables
- **Scope**: Same package + subclasses

```java
```

```java
// File: package1/ProtectedExample.java
package package1;

public class ProtectedExample {
    protected String protectedVariable = "Accessible in package and subclasses";
    protected int protectedNumber = 100;

    protected void protectedMethod() {
        System.out.println("Protected method called");
    }

    protected ProtectedExample() {
        System.out.println("Protected constructor");
    }
}

// File: package1/SamePackageTest.java
package package1;

class SamePackageTest {
    public static void main(String[] args) {
        ProtectedExample obj = new ProtectedExample();    // Accessible - same package
        System.out.println(obj.protectedVariable);        // Accessible - same package
        obj.protectedMethod();                            // Accessible - same package
    }
}

// File: package2/SubclassTest.java
package package2;
import package1.ProtectedExample;

class SubclassTest extends ProtectedExample {
    public SubclassTest() {
        super();                                  // Protected constructor accessible
        System.out.println(protectedVariable);            // Accessible - subclass
        protectedMethod();                                // Accessible - subclass
    }
}

// File: package2/NonSubclassTest.java
package package2;
import package1.ProtectedExample;

class NonSubclassTest {
    public static void main(String[] args) {
        ProtectedExample obj = new ProtectedExample();
```

```
        // System.out.println(obj.protectedVariable);     // Compilation Error - different package, not subclass
        // obj.protectedMethod();                         // Compilation Error
    }
}
```

4. default (Package-Private)

- **Visibility**: Accessible only within the same package
- **Usage**: Classes, methods, constructors, variables
- **Scope**: Same package only
- **Declaration**: No access modifier keyword

java

```java
// File: package1/DefaultExample.java
package package1;

class DefaultExample {                  // Default access for class
    String defaultVariable = "Package private variable";
    int defaultNumber = 50;

    void defaultMethod() {              // Default access for method
        System.out.println("Default method called");
    }

    DefaultExample() {                  // Default access for constructor
        System.out.println("Default constructor");
    }
}

// File: package1/SamePackageAccess.java
package package1;

class SamePackageAccess {
    public static void main(String[] args) {
        DefaultExample obj = new DefaultExample();       // Accessible - same package
        System.out.println(obj.defaultVariable);         // Accessible - same package
        obj.defaultMethod();                             // Accessible - same package
    }
}

// File: package2/DifferentPackageAccess.java
package package2;
// import package1.DefaultExample;                       // Compilation Error - default class not accessible

class DifferentPackageAccess {
    public static void main(String[] args) {
        // DefaultExample obj = new DefaultExample();    // Compilation Error
    }
}
```

Access Modifier Comparison Table

Access Modifier	Same Class	Same Package	Subclass (Different Package)	Different Package
private	✓	✗	✗	✗
default	✓	✓	✗	✗
protected	✓	✓	✓	✗
public	✓	✓	✓	✓

2. Encapsulation in Detail

Encapsulation is the mechanism of wrapping data (variables) and code (methods) together as a single unit, while restricting direct access to some components.

Principles of Encapsulation

1. **Data Hiding**: Make instance variables private
2. **Controlled Access**: Provide public methods to access and modify data
3. **Validation**: Add validation logic in setter methods
4. **Abstraction**: Hide internal implementation details

Benefits of Encapsulation

1. **Data Security**: Prevents unauthorized access
2. **Data Validation**: Ensures data integrity
3. **Flexibility**: Easy to change internal implementation
4. **Maintainability**: Better code organization
5. **Debugging**: Easier to track data changes

Complete Encapsulation Example

```java
```

```java
public class BankAccount {
    // Private instance variables (Data Hiding)
    private String accountNumber;
    private String accountHolderName;
    private double balance;
    private String accountType;
    private boolean isActive;

    // Constructor
    public BankAccount(String accountNumber, String accountHolderName, String accountType) {
        this.accountNumber = accountNumber;
        this.accountHolderName = accountHolderName;
        this.accountType = accountType;
        this.balance = 0.0;
        this.isActive = true;
    }

    // Getter methods (Controlled Read Access)
    public String getAccountNumber() {
        return accountNumber;
    }

    public String getAccountHolderName() {
        return accountHolderName;
    }

    public double getBalance() {
        return isActive ? balance : -1;  // Return -1 if account is inactive
    }

    public String getAccountType() {
        return accountType;
    }

    public boolean isAccountActive() {
        return isActive;
    }

    // Setter methods with validation (Controlled Write Access)
    public void setAccountHolderName(String accountHolderName) {
        if (accountHolderName != null && !accountHolderName.trim().isEmpty()) {
            this.accountHolderName = accountHolderName;
        } else {
            System.out.println("Invalid account holder name");
        }
    }
```

```
public void setAccountType(String accountType) {
    if (accountType.equals("SAVINGS") || accountType.equals("CURRENT") || accountType.equals("FIXED")) {
        this.accountType = accountType;
    } else {
        System.out.println("Invalid account type. Must be SAVINGS, CURRENT, or FIXED");
    }
}

// Business methods with validation
public boolean deposit(double amount) {
    if (!isActive) {
        System.out.println("Cannot deposit. Account is inactive.");
        return false;
    }

    if (amount > 0) {
        balance += amount;
        System.out.println("Deposited: $" + amount + ". New balance: $" + balance);
        return true;
    } else {
        System.out.println("Invalid deposit amount. Must be positive.");
        return false;
    }
}

public boolean withdraw(double amount) {
    if (!isActive) {
        System.out.println("Cannot withdraw. Account is inactive.");
        return false;
    }

    if (amount > 0 && amount <= balance) {
        balance -= amount;
        System.out.println("Withdrawn: $" + amount + ". New balance: $" + balance);
        return true;
    } else {
        System.out.println("Invalid withdrawal amount or insufficient balance.");
        return false;
    }
}

public void deactivateAccount() {
    isActive = false;
    System.out.println("Account has been deactivated.");
}
```

```java
    public void activateAccount() {
        isActive = true;
        System.out.println("Account has been activated.");
    }

    // Private helper method (Internal implementation)
    private void logTransaction(String type, double amount) {
        System.out.println("Transaction Log: " + type + " of $" + amount + " on account " + accountNumber);
    }

    // Method to display account information
    public void displayAccountInfo() {
        System.out.println("=== Account Information ===");
        System.out.println("Account Number: " + accountNumber);
        System.out.println("Account Holder: " + accountHolderName);
        System.out.println("Account Type: " + accountType);
        System.out.println("Balance: $" + (isActive ? balance : "Account Inactive"));
        System.out.println("Status: " + (isActive ? "Active" : "Inactive"));
    }
}
```

3. Getter and Setter Methods in Detail

Getter and setter methods provide controlled access to private variables, implementing the encapsulation principle.

Getter Methods (Accessors)

- **Purpose**: Retrieve the value of private variables
- **Naming Convention**: `get` + PropertyName (camelCase)
- **Return Type**: Same as the variable type
- **Parameters**: Usually no parameters

Setter Methods (Mutators)

- **Purpose**: Modify the value of private variables
- **Naming Convention**: `set` + PropertyName (camelCase)
- **Return Type**: Usually `void` (can return `boolean` for validation status)
- **Parameters**: One parameter of the same type as the variable

Advanced Getter-Setter Examples

1. Basic Getter-Setter Pattern

```java
```

```java
public class Student {
    private String name;
    private int age;
    private String studentId;
    private double gpa;

    // Basic Getters
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public String getStudentId() {
        return studentId;
    }

    public double getGpa() {
        return gpa;
    }

    // Basic Setters
    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setStudentId(String studentId) {
        this.studentId = studentId;
    }

    public void setGpa(double gpa) {
        this.gpa = gpa;
    }
}
```

2. Getter-Setter with Validation

```java
```

```java
public class Employee {
    private String name;
    private int age;
    private double salary;
    private String email;
    private String department;

    // Getters
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public double getSalary() {
        return salary;
    }

    public String getEmail() {
        return email;
    }

    public String getDepartment() {
        return department;
    }

    // Setters with Validation
    public boolean setName(String name) {
        if (name != null && !name.trim().isEmpty() && name.length() >= 2) {
            this.name = name;
            return true;
        } else {
            System.out.println("Invalid name. Name must be at least 2 characters long.");
            return false;
        }
    }

    public boolean setAge(int age) {
        if (age >= 18 && age <= 65) {
            this.age = age;
            return true;
        } else {
            System.out.println("Invalid age. Age must be between 18 and 65.");
            return false;
```

```
        }
    }

    public boolean setSalary(double salary) {
        if (salary > 0) {
            this.salary = salary;
            return true;
        } else {
            System.out.println("Invalid salary. Salary must be positive.");
            return false;
        }
    }

    public boolean setEmail(String email) {
        if (email != null && email.contains("@") && email.contains(".")) {
            this.email = email;
            return true;
        } else {
            System.out.println("Invalid email format.");
            return false;
        }
    }

    public boolean setDepartment(String department) {
        String[] validDepartments = {"IT", "HR", "FINANCE", "MARKETING", "OPERATIONS"};
        for (String dept : validDepartments) {
            if (dept.equals(department.toUpperCase())) {
                this.department = department.toUpperCase();
                return true;
            }
        }
        System.out.println("Invalid department. Must be one of: IT, HR, FINANCE, MARKETING, OPERATIONS");
        return false;
    }
}
```

3. Advanced Getter-Setter Patterns

java

```
public class Product {
    private String productId;
    private String productName;
    private double price;
    private int quantity;
    private boolean isAvailable;
    private String category;

    // Read-only property (only getter)
    public String getProductId() {
        return productId;
    }

    // Write-only property (only setter) - rare but sometimes useful
    public void setInternalCode(String code) {
        // Some internal processing
        System.out.println("Internal code set: " + code);
    }

    // Computed property (getter that calculates value)
    public double getTotalValue() {
        return price * quantity;
    }

    // Getter with transformation
    public String getFormattedPrice() {
        return String.format("$%.2f", price);
    }

    // Getter with conditional logic
    public String getAvailabilityStatus() {
        if (isAvailable && quantity > 0) {
            return "In Stock";
        } else if (isAvailable && quantity == 0) {
            return "Out of Stock";
        } else {
            return "Discontinued";
        }
    }

    // Setter with side effects
    public void setPrice(double price) {
        if (price > 0) {
            this.price = price;
            // Side effect: update availability based on price
            if (price > 1000) {
```

```
            System.out.println("High-value item flagged for special handling");
        }
    }
}

// Fluent setter (returns this for method chaining)
public Product setProductName(String productName) {
    if (productName != null && !productName.trim().isEmpty()) {
        this.productName = productName;
    }
    return this;  // Enable method chaining
}

public Product setQuantity(int quantity) {
    if (quantity >= 0) {
        this.quantity = quantity;
        this.isAvailable = quantity > 0;
    }
    return this;
}

public Product setCategory(String category) {
    if (category != null) {
        this.category = category.toUpperCase();
    }
    return this;
}
}

// Usage of fluent setters
class TestFluent {
public static void main(String[] args) {
    Product product = new Product()
        .setProductName("Laptop")
        .setQuantity(10)
        .setCategory("electronics");
}
}
```

Best Practices for Getter-Setter Methods

1. **Always validate in setters**: Check for null values, range validation, format validation
2. **Return boolean from setters**: Indicate success/failure of the operation
3. **Use meaningful error messages**: Help users understand what went wrong
4. **Consider immutable objects**: For some cases, consider making objects immutable

5. **Avoid exposing internal collections directly**: Return copies instead
6. **Use proper naming conventions**: `get`/`set` prefix with proper camelCase
7. **Consider lazy initialization**: Initialize expensive objects only when needed

Complete Example with All Concepts

java

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Date;

public class CompleteEncapsulationExample {
    // Private fields (Data Hiding)
    private String name;
    private int age;
    private List<String> hobbies;
    private Date createdDate;

    // Constructor
    public CompleteEncapsulationExample(String name) {
        this.name = name;
        this.hobbies = new ArrayList<>();
        this.createdDate = new Date();
    }

    // Getter with validation
    public String getName() {
        return name != null ? name : "Unknown";
    }

    // Setter with validation
    public boolean setName(String name) {
        if (name != null && name.trim().length() > 0) {
            this.name = name.trim();
            return true;
        }
        return false;
    }

    // Getter for age
    public int getAge() {
        return age;
    }

    // Setter with range validation
    public boolean setAge(int age) {
        if (age >= 0 && age <= 150) {
            this.age = age;
            return true;
        }
        System.out.println("Invalid age: " + age);
        return false;
    }
```

```
    // Getter that returns copy of list (prevents external modification)
    public List<String> getHobbies() {
        return new ArrayList<>(hobbies);
    }

    // Method to add hobby with validation
    public boolean addHobby(String hobby) {
        if (hobby != null && !hobby.trim().isEmpty() && !hobbies.contains(hobby)) {
            hobbies.add(hobby);
            return true;
        }
        return false;
    }

    // Method to remove hobby
    public boolean removeHobby(String hobby) {
        return hobbies.remove(hobby);
    }

    // Read-only getter for creation date
    public Date getCreatedDate() {
        return new Date(createdDate.getTime());  // Return copy to prevent modification
    }

    // Display method
    public void displayInfo() {
        System.out.println("Name: " + getName());
        System.out.println("Age: " + getAge());
        System.out.println("Hobbies: " + getHobbies());
        System.out.println("Created: " + getCreatedDate());
    }

    public static void main(String[] args) {
        CompleteEncapsulationExample person = new CompleteEncapsulationExample("John Doe");

        person.setAge(25);
        person.addHobby("Reading");
        person.addHobby("Swimming");

        person.displayInfo();

        // Attempting to modify returned list won't affect original
        List<String> hobbies = person.getHobbies();
        hobbies.add("Hacking");  // This won't affect the original hobbies list

        System.out.println("\nAfter attempting to modify returned list:");
```

```
        person.displayInfo();
    }
}
```

Key Takeaways

1. **Access Modifiers** provide different levels of visibility control
2. **Encapsulation** combines data and methods while controlling access
3. **Getter-Setter** methods provide controlled access to private data
4. **Validation** in setters ensures data integrity
5. **Proper encapsulation** leads to more secure and maintainable code

These concepts work together to create robust, secure, and maintainable Java applications.